

# Game Structure Overview

## 1 Moteur de jeu

Le **Moteur de jeu** est le composant centrale qui gère le boucle de jeu et coordonne les composant de la structure MVC, Modèle, View, Controlleur.

- Boucle de jeu, et son arrêt.
- Crée composant MVC et les lie entre eux.

```
class GameEngine
    render = new Renderer()
    gameState = new GameStateHandler()
    controller = new Controller()
    fun run()
        while (running)
            processInput()
            updateGame()
            refresh()
        end run
    end GameEngine
```

## 2 Model

Le **Model** represente le noyau de la logic de jeu, pour que le controlleur ai acces au composant du model, on utilise une class de type facade StateManager.

### 2.1 Level

Level, class de type container, contient tout les elements de jeu (Palette, Brick, TextContainer, etc). Methode all(), renvoie un std::tuple contenant tout les elements du level. Utiliser std::apply pour iterer sur le tuple.  
<https://stackoverflow.com/questions/1198260/how-can-you-iterate-over-the-elements-of-an-stdtuple>

### 2.2 StateManager

Contient les methodes pour modifier les composant de level, contient un pointeur vers level.

```

class Paddle
    int pos
    moveLeft(dx, dy)
end class
class StateManager
    movePaddleRight()
    ...
end class
class Level
    Paddle
    ...
end class

```

### 3 Contrôleur

Contient un StateManager, Level et Renderer, recupere le input utilisateur, modifie les objets de jeu et render le resultat a chaque frame

```

class Controller {
    int getInput() {
        if (keyPressed(LEFT)) return LEFT;
        if (keyPressed(RIGHT)) return RIGHT;
    }
}

```

### 4 Inheritance

Inheritance is a core concept in object-oriented programming where a class (child) derives from another class (parent) and inherits its properties and methods. This establishes an "is-a" relationship between the child and parent classes. However, inheritance can lead to tightly coupled code and deep hierarchies, making the system harder to maintain and extend.

```

class Animal {
public:
    virtual void sound() = 0;
};

class Dog : public Animal {
public:
    void sound() override { std::cout << "Bark"; }
};

```

### 5 Composition

Composition is a design principle where objects are composed of other objects to achieve complex functionality. This follows the "has-a" relationship, where one class contains another. Composition provides more flexibility, as components can be easily swapped or modified at runtime without modifying the entire class structure.

```
class SoundBehavior {
public:
    virtual void sound() = 0;
};

class BarkBehavior : public SoundBehavior {
public:
    void sound() override { std::cout << "Bark"; }
};

class Dog {
private:
    SoundBehavior* soundBehavior;
public:
    Dog(SoundBehavior* sb) : soundBehavior(sb) {}
    void makeSound() { soundBehavior->sound(); }
};
```